Pay Attention to Racket

by Anthony Carrico <acarrico@memebeam.org>

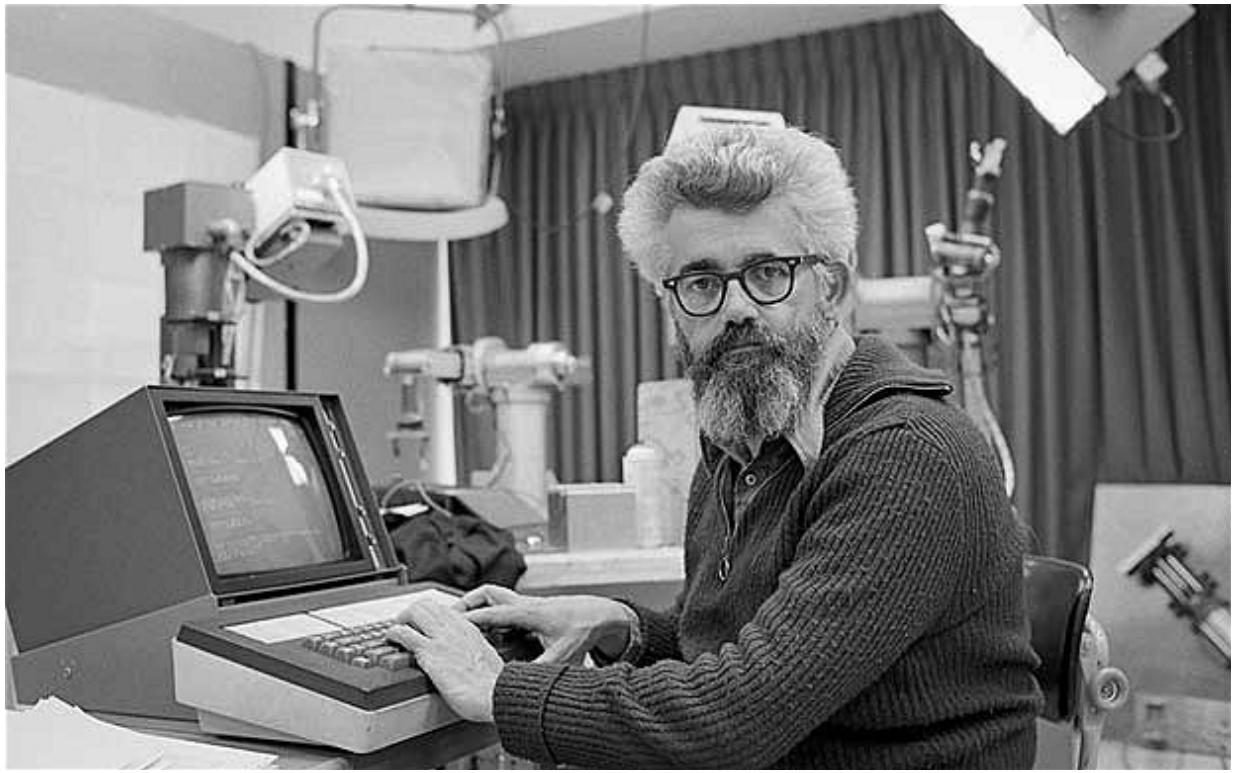
@Anthony_Carrico

https://github.com/acarrico/

#vtfun @racketlang

Vermont Functional Users Group

John McCarthy Red Sox Beard Award



Revised Report on the Algorithmic Language Algol 60

By J.W. Backus, F.L. Bauer, J.Green, C. Katz, J. McCarthy, P. Naur, A.J. Perlis, H. Rutishauser, K. Samuelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden, M. Woodger, Edited by Peter Naur

Revised(3) Report on the Algorithmic Language Scheme

Dedicated to the Memory of ALGOL 60

By H. Abelson, R. K. Dybvig, C. T. Haynes,G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. J. Sussman, D. H. Bartley, R. Halstead, D. Oxley, M. Wand, G. Brooks, C. Hanson, K. M. Pitman, Edited by Jonathan Rees and William Clinger

Mathias Felleisen, "Racket is ..."

- a programming language

Mathias Felleisen, "Racket is ..."

- a programming language
- a family of programming languages

racket/base

"reminiscent of plain Scheme"

Mathias Felleisen, "Racket is ..."

- a programming language
- a family of programming languages

racket/base racket

"classes, mixins, and traits", and much more

Mathias Felleisen, "Racket is ..."

- a programming language
- a family of programming languages

racket/base racket typed/racket

"as type-safe as ML"

From Scripts to Programs

"Modules + contracts + occurence typing enables programmers to enrich existing Scheme-style code with types, while requiring few changes."

Mathias Felleisen, "Racket is ..."

- a programming language
- a family of programming languages

racket/base racket typed/racket frtime

dataflow reactive frp

Mathias Felleisen, "Racket is ..."

- a programming language
- a family of programming languages

  racket/base racket typed/racket frtime lazy

  "stream-based programming like Haskell"

Mathias Felleisen, "Racket is ..."

- a programming language
- a family of programming languages

racket/base racket typed/racket frtime lazy web-server

Mathias Felleisen, "Racket is ..."

- a programming language
- a family of programming languages

racket/base racket typed/racket frtime lazy web-server

scribble

Mathias Felleisen, "Racket is ..."

- a programming language
- a family of programming languages

racket/base racket typed/racket frtime lazy web-server

scribble redex

Mathias Felleisen, "Racket is ..."

- a programming language
- a family of programming languages

racket/base racket typed/racket frtime lazy web-server

scribble redex datalog

Mathias Felleisen, "Racket is ..."

- a programming language
- a family of programming languages

racket/base racket typed/racket frtime lazy web-server

scribble redex datalog racklog

Mathias Felleisen, "Racket is ..."

- a programming language
- a family of programming languages

racket/base racket typed/racket frtime lazy web-server

scribble redex datalog racklog slideshow

Mathias Felleisen, "Racket is ..."

- a programming language
- a family of programming languages

racket/base racket typed/racket frtime lazy web-server

scribble redex datalog racklog slideshow more

Mathias Felleisen, "Racket is ..."

- a programming language
- a family of programming languages

racket/base racket typed/racket frtime lazy web-server

scribble redex datalog racklog slideshow more

- a programming language for programming languages

"linguistic mechanisms that enable the quick construction of reliable languages, language fragments, and their composition."

Slideshow

```
#lang slideshow

(require slideshow/code)
(require "helpers.rkt")
(slide
 #:title "Pay Attention to Racket"
 (t "by Anthony Carrico <acarrico@memebeam.org>")
 (t "@Anthony_Carrico")
 (t "https://github.com/acarrico/")
 (bitmap "functional.png")
 (t "#vtfun @racketlang")
 (t "Vermont Functional Users Group"))
```

Slideshow

```
#lang slideshow

(require slideshow/code)
(provide lang foreign-code algol60-code highlight-la
  cite item-cite colorize-words blue red yellow anor
(define-syntax-rule (lang l)
  (vl-append
   (hbl-append (tt "#lang ") (code l))
   (blank (+ ((get-current-code-font-size))
             (current-code-line-sep)))))
(define (foreign-code str)
  (apply
   vl-append
   (current-code-line-sep)
   (map (lambda (str)
          (para #:fill? #t (tt str)))
        (string-split str "\n"))))
```

Brainf*ck

```
#lang planet dyoo/bf
++++++ [>++++++++++++<-] >.
>++++++++++ [>++++++++++<-] >+.
+++++++..+++.>++++ [>++++++++++<-] >.
<+++ [>----<-] >.<<<<+++ [>+++++<-] >.
>>.+++.------.--------.>>+.
```

Brainf*ck

```
#lang planet dyoo/bf
++++++ [>++++++++++++<-] >.
>++++++++++ [>++++++++++<-] >+.
+++++++..+++.>++++ [>++++++++++<-] >.
<+++ [>----<-] >.<<<<<+++ [>+++++<-] >.
>>.+++.------.--------.>>+.
```

```
Hello, World!
```

Datalog

```
#lang datalog
ancestor(A, B) :- parent(A, B).
ancestor(A, B) :-
parent(A, C), D = C, ancestor(D, B).
parent(john, douglas).
parent(bob, john).
ancestor(A, B)?
```

Datalog

```
#lang datalog
ancestor(A, B) :- parent(A, B).
ancestor(A, B) :-
parent(A, C), D = C, ancestor(D, B).
parent(john, douglas).
parent(bob, john).
ancestor(A, B)?
```

```
ancestor(bob, john).
ancestor(john, douglas).
ancestor(bob, douglas).
```

Algol60

```
#lang algol60

begin
  integer procedure FACTORIAL(n);
    value n;
    integer n;
    FACTORIAL :=
      if (n < 2)
      then 1
      else n * FACTORIAL(n-1);
```

Algol60

```
  printnln(FACTORIAL(1));
  printnln(FACTORIAL(2));
  printnln(FACTORIAL(3));
  printnln(FACTORIAL(40));
end
```

Algol60

```
  printnln(FACTORIAL(1));
  printnln(FACTORIAL(2));
  printnln(FACTORIAL(3));
  printnln(FACTORIAL(40));
end
```

```
1
2
6
265252859812191058636308480000000
```

List Processing

1956. Dartmouth Summer Research Project.

John McCarthy is thinking about:

- advice taker (proto logic programming)
- symbolic differentiation (proto functional programming)

List Processing

1956. Dartmouth Summer Research Project.

Newell, Shaw, and Simon present IPL 2, an assembly language for list processing.

Instead of just working with numbers, and arrays of numbers, let's compute with recursive data structures!

List Processing

recursive data structures + algebraic notation

Meanwhile IBM's Fortran project introduces the idea of writing programs algebraically.

How? Add it to Fortran?

1957. John McCarthy experiments with a combination of FORTRAN and list processing.

1957. John McCarthy meets Steve Russell.

Algol58? A new language?

John McCarthy fails to push this stuff into Algol58.

Expressions

"The convenience of algebraic notation for the description of procedures derives in large part from the fact that frequently the output of one procedure serves the input for another and the functional notation enables us to avoid giving the intermediate result a name."

John McCarthy. *An Algebraic Language for the Manipulation of Symbolic Expressions*. MIT AI Lab Memo 1. September, 1958.

Conditional Expressions

$$(p_1 \rightarrow e_1, \cdots, p_n \rightarrow e_n)$$

1957 John McCarthy is frustrated with IF procedures and IF statements.

"One of the weakest features of present programming languages is in the means they provide for treating conditionals, that is the calculation of quantities where the operations used depends on the result of certain prior tests."

John McCarthy. *An Algebraic Language for the Manipulation of Symbolic Expressions*. MIT AI Lab Memo 1. September, 1958.

Conditional Expressions

$$(p_1 \rightarrow e_1, \cdots, p_n \rightarrow e_n)$$

"The use of propositional quantities, predicates, and conditional expressions, essentially solves this problem and eliminates most of the small scale branching from source programs."

John McCarthy. *An Algebraic Language for the Manipulation of Symbolic Expressions*. MIT AI Lab Memo 1. September, 1958.

Conditional Expressions

$$(p_1 \rightarrow e_1, \cdots, p_n \rightarrow e_n)$$

"A paper defining conditional expressions and proposing their use in Algol was sent to the CACM, but was demoted to a letter to the editor, because it was very short."

Maybe??? John McCarthy. *Letter to the Editor*. CACM, Vol. 2, No. 8. 1959.

The ACM subsequently apologized.

Footnote in the modern typeset version of his big paper. Also in LISP History.

Conditional Expressions

$$|x| = (x < 0 \rightarrow -x, T \rightarrow x)$$

$$\delta_{ij} = (i = j \rightarrow 1, T \rightarrow 0)$$

$$sgn(x) = (x < 0 \rightarrow -1, x = 0 \rightarrow 0, T \rightarrow 1)$$

John McCarthy. *Recursive Functions of Symbolic Expressions and Their Computation by Machine*. CACM. April, 1960.

Interlude: Obvious Right?

1958

- List processing AKA recursive data structures.
- Algebraic notation AKA expressions.
- Conditional expressions AKA branching problem solved.

Good! Good! Good! Conclusion:

- Programming languages will start with this base.

NOTE: No esoteric compiler or runtime technology required (yet).

Interlude: Wrong

"Some corporate programming guidelines list the use of the conditional operator as bad practice because it can harm readability and long-term maintainability."

?: in style guidelines. Wikipedia. October, 2013.

"?:" or "ternary operator" is what imperative programmers call conditional expressions.

Interlude: Wrong

over 30 years later...

Python version 0.9.0 was released in February 1991.

finally, almost 50 years later...

Python version 2.5 was released in September 2006 with conditional expressions.

x = true_value if condition else false_value

but imagine nesting that syntax!

Interlude: Wrong

over 50 years later:

"Q: Does Go have the ?: operator?". The Go Programming Language FAQ. October, 2013.

"A: There is no ternary form in Go. You may use the following to achieve the same result:"

```
if expr {
    n = trueVal
} else {
    n = falseVal
}
```

Go you are making me cry! That is not a conditional expression.

Recursion

"The use of propositional quantities, predicates, and conditional expressions, essentially solves this problem and eliminates most of the small scale branching from source programs.

In combination with the feature of recursive definition it permits certain subroutines to be defined by single formulas in this language that are quite involved programs in other languages."

John McCarthy. *An Algebraic Language for the Manipulation of Symbolic Expressions*. MIT AI Lab Memo 1. September, 1958.

Recursion

$$n! = (n = 0 \rightarrow 1, T \rightarrow n \cdot (n-1)!)$$

John McCarthy. *Recursive Functions of Symbolic Expressions and Their Computation by Machine*. CACM. April, 1960.

He also shows one liners for GCD and SQRT.

The **n**'s are all over the place. What does this mean?

Higher-Order Functions

3.3.4. Maplist (L,J,f(J)). The value of this function is the address of a list formed from the list L by mapping the element J into f(J).

John McCarthy. *An Algebraic Language for the Manipulation of Symbolic Expressions*. MIT AI Lab Memo 1. September, 1958.

There is a problem here with the **J**.

Incidentally, this can be made to work in Algol60.

Higher-Order Functions

```
#lang algol60
begin
  integer procedure SIGMA(x, i, n);
    value n; integer x, i, n;
  begin
    integer sum; sum := 0;
    for i := 1 step 1 until n do
      sum := sum + x;
    SIGMA := sum;
  end;
  integer q;
  printnln(SIGMA(q*2-1, q, 7));
end
```

49

Functions and Forms

"It is usual in mathematics—outside of mathematical logic—to use the word “function” imprecisely and to apply it to forms such as **`y^2 + x`**. Because we shall later compute with expressions for functions, we need a distinction between functions and forms and a notation for expressing this distinction."

John McCarthy. *Recursive Functions of Symbolic Expressions and Their Computation by Machine*. CACM. April, 1960.

Functions and Forms

"Let f be an expression that stands for a function of two integer variables. It should make sense to write `f(3, 4)` and the value of this expression should be determined. The expression `y^2 + x` does not meet this requirement; `y^2 + x(3, 4)` is not a conventional notation, and if we attempted to define it we would be uncertain whether its value would turn out to be 13 or 19."

John McCarthy. *Recursive Functions of Symbolic Expressions and Their Computation by Machine*. CACM. April, 1960.

Functions and Forms

"Church calls an expression like **`y^2 + x`**, a form. A form can be converted into a function if we can determine the correspondence between the variables occurring in the form and the ordered list of arguments of the desired function. This is accomplished by Church's λ-notation."

John McCarthy. *Recursive Functions of Symbolic Expressions and Their Computation by Machine*. CACM. April, 1960.

Recursion

```
#lang racket

(define (! n)
  (if (< n 2) 1 (* n (! (- n 1)))))
```

just syntactic sugar for this:

```
(define !
  (lambda (n)
    (if (< n 2) 1 (* n (! (- n 1))))))
```

Recursion

```
#lang algol60

begin
  integer procedure FACTORIAL(n);
    value n;
    integer n;
    FACTORIAL :=
      if (n < 2)
      then 1
      else n * FACTORIAL(n-1);
end
```

Higher-Order Functions

$$maplist[x; f] =$$
$$[null[x] \rightarrow NIL; T \rightarrow cons[f[x]; maplist[cdr[x]; f]]]$$

John McCarthy. *Recursive Functions of Symbolic Expressions and Their Computation by Machine*. CACM. April, 1960.

NOTE: Back in the day, they called these functional arguments or FUNARGs.

Higher-Order Functions

```
#lang "racket"

(define (maplist x f)
  (if (pair? x)
      (cons (f x)
            (maplist (cdr x) f))
      '()))

(maplist '("a" "b" "c") length)
```

```
'(3 2 1)
```

Maplist is a weird function, so let's skip it and look at **map**.

Higher-Order Functions

```
#lang "racket"

(define (map f lst)
  (if (pair? lst)
      (cons (f (car lst))
            (map f (cdr lst)))
      '()))

(map string-upcase '("a" "b" "c"))
```

```
'("A" "B" "C")
```

A slightly beefier version of **`map`** is built into Scheme and Racket.

Conclusion: Obvious Right?

1960

- List processing AKA recursive data structures.
- Algebraic notation AKA expressions.
- Conditional expressions AKA branching problem solved.
- Recursion.
- Funargs AKA higer order functions.
- Lambda.

Good! Good! Good! Conclusion:

- Programming languages will start with this base.

Loose Ends in this Narrative

- Steve Russell
- Universal Function
- John McCarthy. *Recursive Functions of Symbolic Expressions and Their Computation by Machine*. CACM. April, 1960.
- Lisp and Algol60
- REQUEST: P. J. Landin The Next 700 Programming Languages. CACM 9(3), March 1966. SASL, Miranda, ML, Haskell, Clean, Lucid. "Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things."

Part 2 – Lambda the Ultimate!

Today I wanted to tell you the story of Scheme and Racket, but first I had to tell you the story of Lisp and Algol60.

sorry

Promo

This is VTFun's first Racket or Scheme talk, but instead of a tutorial, I'm planning a tour of a few key ideas from the 1950s to the present that every programmer should grok, and no language designer should ignore. I hope that each slide will be easy to follow, seem obvious in retrospect, and maybe make you a better programmer.

Bio

About the Presenter: Anthony Carrico earned a master's degree in electrical and computer engineering, primarily in signal processing and computer architecture. He tried to learn to code at an AI company heated with Lisp and Unix machines. He went on to program Japanese music video games. His wife teaches at St. Mike's and he stays home with the kids.

Themes

- Lambda is abstraction. The primitives make it functional, actor, oop, relational, constraint, or xxxxxx style (to paraphrase Steele).
- Understand Xxx Style in terms of abstraction over primitives before adding sugar. (Hmm, I'll credit Friedman this time).
- Worse may be better in the short term, but pay attention to the communities that cultivate best practices over the long term.
- The turtle keeps on going long after the hare is dead, decomposed, and forgotten.

fin